# RECONFIGURING THE STRUCTURE OF COMPONENT-BASED SYSTEMS

Martin TOMÁŠEK*
*Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55 602 3178, e-mail: martin.tomasek@tuke.sk

**ABSTRACT**

*We introduce component-based system architecture and basic software components. The main goal of this paper is to present a novel solution for component-based system design in UML language. Any UML diagram of component-based system is automatically optimized and then reconfigured to satisfy the hierarchical structure of component-based system. This solution is created as a plug-in for Eclipse IDE and the results of the work are tested in various practical examples.*

***Keywords:*** *software component, component-based system, reconfiguration*

## 1. INTRODUCTION

Idea, that software should be created from components, composed with previously developed parts, is not new. This idea was presented for the first time at NATO conference about software engineering in 1968 [1]. Component-based software development approach in software designing is based on repeated usage of objects that are called software components. Component-based software engineering was established because object oriented software design was not effective in supporting objects reusability. This fail of object-oriented programing came because class objects are in general very detailed and specific. Software components are more abstract than objects and can provided services for other components, therefore they should exist as separate entities.

The goal of this paper is to find a solution for design and optimization of component systems structure followed by the implementation. Finally, the solution is implemented as a software design tool and the functionality of this solution is tested in different use-cases.

## 2. COMPONENT-BASED SYSTEM

Component-based development or component-based programing is modular extension of object-oriented programing. Component-based programing uses all equipment of object-oriented programing with objects created components, which are software units with higher quality level. The basic principle of component-based programing is in creation of capable components, which are capable enough to be able to take responsibility for the implementation of a set of activities of related applications or computer systems. Component-based programing is based on a prediction that software should be created the same way as manufacture products that are manufactured in mass production belt [2].

Components provide a service without regard to where the component is executing or its programming language. A component is an independent executable entity that can be made up of one or more executable objects. The component interface is published and all interactions are performed through the published interface [3].

Component application is made from one or more components. Each component is responsible an implementation of an application operations set. Thus we can say that the final version of the application may be highly variable and thus configurable. Because the components are designed on a modular base, we can create diverse collection of components. The modularity of components has advantages not only for programmers but also for users. Depending on user's request, customer can directly influence the final composition of component applications.

### 2.1. Properties of Components

Each component must fulfill following features:

- **Identification**
  The component must be clearly identifiable. For clearly identification of the components contributes certification of these components and also increasing the credibility of the component. Component certification is the process to control that the component satisfies their specification. Certification means that someone other except programmer can controls quality of the component.

- **Encapsulation**
  The component encapsulates all the functionality inside. Inside the component there is a set of objects with delegated responsibilities for implementation of specific activities. With these objects, components communicate by means of mechanism of sending and receiving messages.

- **Interface**
  Components also have their own public access interface and users can use their services. Component interfaces are divided into two groups:

  1. Provided interface defines services that are provided by other components. This interface defines methods of component that can be called by user. It's usually component of API.
  2. Required interface defines the services that must be available to enable component to execute its activity.

- **Preparations for use**
  The component operates as a ready software element that provides comprehensive automation of selected

activities. Assuming that the client has an access to a component, it can immediately start using the services provided through its public access interface.

- **Reusability**
  When the component is developed, debugged, tested and optimized, we can use it as many times as it is necessary.

- **Anonymity of users**
  The functionality of component is completely separated from the application. The component must provide quality service in a short time for every of requesting users.

- **Interoperability**
  Components must be able to cooperate with others, even when they are created in various integrated development environments and programming languages.

### 2.2. Aggregation and Composition of Components

Some components may create units, which themselves can become components. Assembling the components to create a functional system is called composition. This composition includes the integration of components with each other, it also consists of integration with the infrastructure component. Type composition between components:

- Sequential composition where the composed components are executed in sequence. This involves composing the provided interfaces of each component.

- Hierarchical composition where one component calls services of another. The provided interface of one component is composed with the required interface of another one.

- Additive composition where the interfaces of two components are put together to create a new component.

## 3. RECONFIGURATION OF THE COMPONENTS

One of the main problems of the component-based systems engineering is to find appropriate notations for describing the systems. With formalized notation it is possible to document component-based designs clearly, automate their analysis and generate system.

The simplest and currently the most widely using model for the mapping of software components is object modeling notation [4]. The most suitable description of object modeling notation is UML language. This description is clear for users and allows them the partial support of the implementation (class generation) in many software products, that support UML language [5].

### 3.1. Language Describing Component-Based Systems

At first we defined language for component description. We decided that the best way is to create new language that describe both component-based system and every component in the system. Syntax and semantic of the language is following:

| **Syntax** | **Semantic** |
|---|---|
| START | begin of component-based system |
| END | end of component-based system |
| COMPONENT | begin of component |
| ENDOFCOMPONENT | end of component |
| NAME | unique component name |
| INPUTTYPE | type(s) of input (required) interfaces of component |
| OUTPUTTYPE | type(s) of output (provided) interfaces of component |
| NEXT | name(s) of next component in component-based system |
| PACKAGE | when are placed in editor package figure, name of package which is the component |
| FIRSTCOMPONENT | when are placed in editor sign for begin reconfiguration, name of first component after this sign |
| LASTCOMPONENT | when are placed in editor sign for end recofiguation, name of last component before this sign |

Schema of language for component-based system description is displayed on Fig. 1.

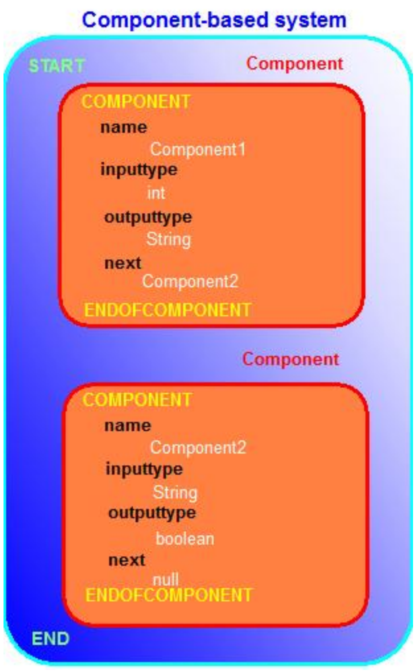

**Fig. 1** Language describing component-based systems.

### 3.2. Reconfiguration Algorithm

Algorithm for reconfiguration of components in the component-based system to satisfy the good hierarchical structure created by the elements:

1. At the beginning of algorithm the component is founded. This component does not have next component or is marked by end sign.
2. After the element is found, there is a check if provided interfaces of previous component are in hierarchical structure with required interfaces of this component.
3. When these interfaces have good hierarchical structure, this component is removed from next component, in every component of the component-based system.
4. This component is removed from list of components of component-based system. The next one is found if there is no next component. This is repeated until all of the components are verified.
5. At the end, the list is returned with appropriate hierarchical structure. Java project is created from this output.

## 4. TOOL FOR RECONFIGURATION OS COMPONENT-BASED SYSTEM STRUCTURE

We have created a plug-in for the Eclipse framework to design and reconfigure component-based system structure. The tool creates Java project from the design when the system complies with given hierarchical structure. Type of the plug-in is multi-page editor view. First page contains designed component-based systems using graphical UML components. On the second page there are supported types for required and provided interfaces. And on the last page is preview of designed system in the XML language. Editor on first page contains palette which is divided in four sections:

1. **Selection** - This item contains elements for marking each graphical component.
2. **Common** - This item contains elements for marking each graphical component.
3. **Entity** - This item contains basic graphical elements for creating component-based system.
4. **Relation** - This item contains element for connecting graphical elements.

Modular composition of the plug-in with the entry in the UML is shown in Fig 2 and consists of:

- **Input** - Component-based system described with UML diagrams.
- **Source codes and files** - Source codes of plug-ins and pictures used for this plug-in.
- **Properties files** - In these files there is a definition of the language syntax for description of component-based systems and syntax of Java language for generation of the component project output.
- **Eclipse modules** - Modules of Eclipse developed environment that our plug-in needs to work.

- **XML serialization/deserialization** - The module for serialization/deserialization of objects in XML language used for saving and opening files.

- **Output Java project** - After successful verification and optimization the output of the plug-in generates Java Eclipse project with desired interfaces.

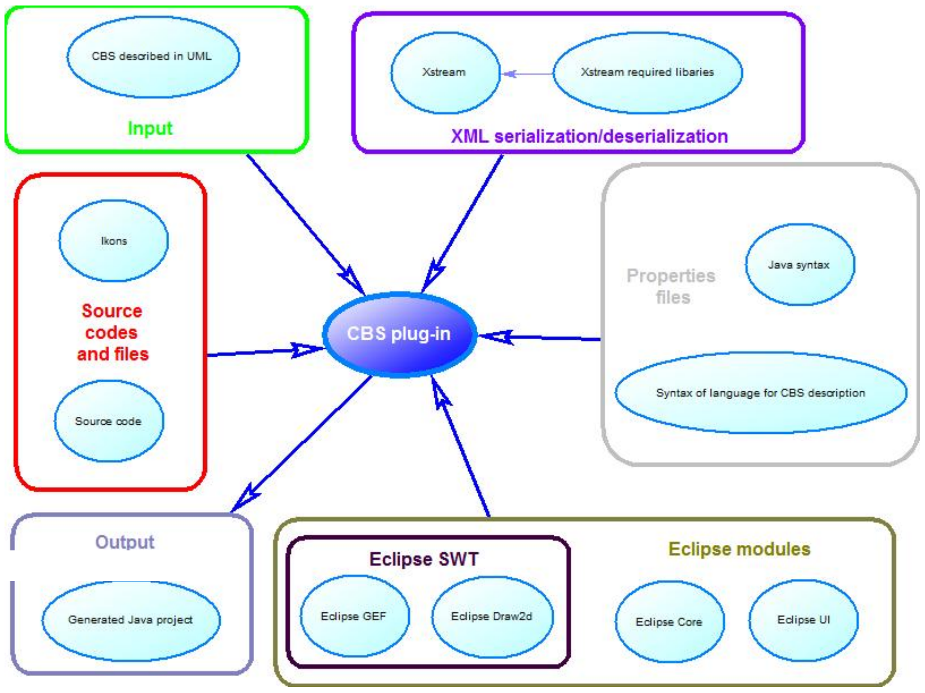

**Fig. 2** The composition of the plug-in with UML.

The internal structure of the plug-in (Fig. 3) providing the reconfiguration to satisfy the hierarchical structure, consists of the following parts:

- **Entrance Control**
  It checks whether each element in component-based system contains closed sign of the beginning and end of the component. Every component-based system is enclosed by sign for the beginning and end of the component system.

- **Syntax checker**
  It checks whether component has correct structure, described types of input and output interfaces and if it contains the unique name in the component systems, name and position of the next component in the component systems.

- **Creating components**
  After input control, the creation of a structures of individual components in the component systems follows.

- **Finding unused components**
  In this part we are trying to discover unused component. Unused component is component, whose name is not used as next component name. This control does not apply on the first component in a system, that input (required) interfaces are the input interface of a component system.

- **Testing hierarchical structure of component-based system**
  After finding unused components, the system creates a hierarchical structure of components without unused components. In this structure it is tested that designed component-based systems complies with original structure.

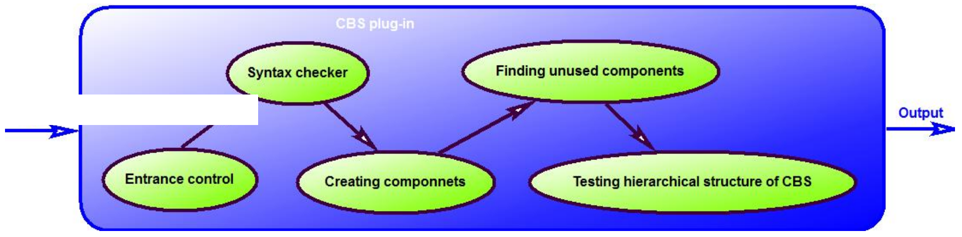

**Fig. 3** The internal structure of the plug-in.

## 5. CONCLUSIONS

The tool that has been created is successful in reconfiguration of the component-based systems. It removes unused components and verifies whether system has good hierarchical structure of component system. After successful verification it is possible to create Java project from the design. This possibility is default in order, because every program should have some output. To test the results we used approach found in [6].

Actual version of program can create project only in Java language. The plan is to make some changes and add support for more object-oriented programing languages as C# and Python.

Next function that can be added is reverse engineering so that final component-based system does not have to be drawn in UML.

This program is only for Eclipse, it could be some kind of disadvantage in the future. So it is good plan to create plugin for other IDEs.

## ACKNOWLEDGEMENT

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10.

## REFERENCES

[1] McIIROY, M. D.: Mass produced software components. Software engineering, NATO Conference Garmisch, 1968.

[2] DHAMI, H. P. S.: Composability of Components in Component-Based Software Development. International Journal of Enterprise Computing and Business Systems, 2012. Brno, 2008. pp. 24–30.

[3] SOMMERVILLE, I.: Software Engineering (8th Edition). University of St Andrews, St Andrews, 2007, pp. 439–461.

[4] LEAVENS, T. G. – SITARAMAN, M.: Foundations of Component-Based Systems. Cambridge University Press, 2000, pp. 47–48.

[5] HEINEMAN, G. – CRNKOVIC, I. – SCHMIDT, W. H. – STAFFORD, A. J. – SZYPERSKI, C. – WALLNAU, K.: Component-Based Software Engineering, 8th International Symposium, CBSE 2005.

[6] GAO, Z. J. – TSAO, H. S. J. – WU, Y.: Testing and Quality Assurance for Component-Based Software, Artech House Publishers, 2003.

Received January 14, 2014, accepted April 22, 2014

## BIOGRAPHY

**Martin Tomášek** received the master degree in computer science in 1998 and PhD degree in software and information systems in 2005 both at the Faculty of Electrical Engineering and Informatics of the Technical University of Koice, Slovakia. Currently he is an associate professor at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics of the Technical University of Koice, Slovakia. His research interests include distributed systems, component-based systems, and concurrency theory.